Thread Coordination -Managing Concurrency

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 8

Sept 17, 2014

https://computing.llnl.gov/tutorials/pthreads/

Reading: A&D 5-5.6
HW 2 out
Proj 1 out

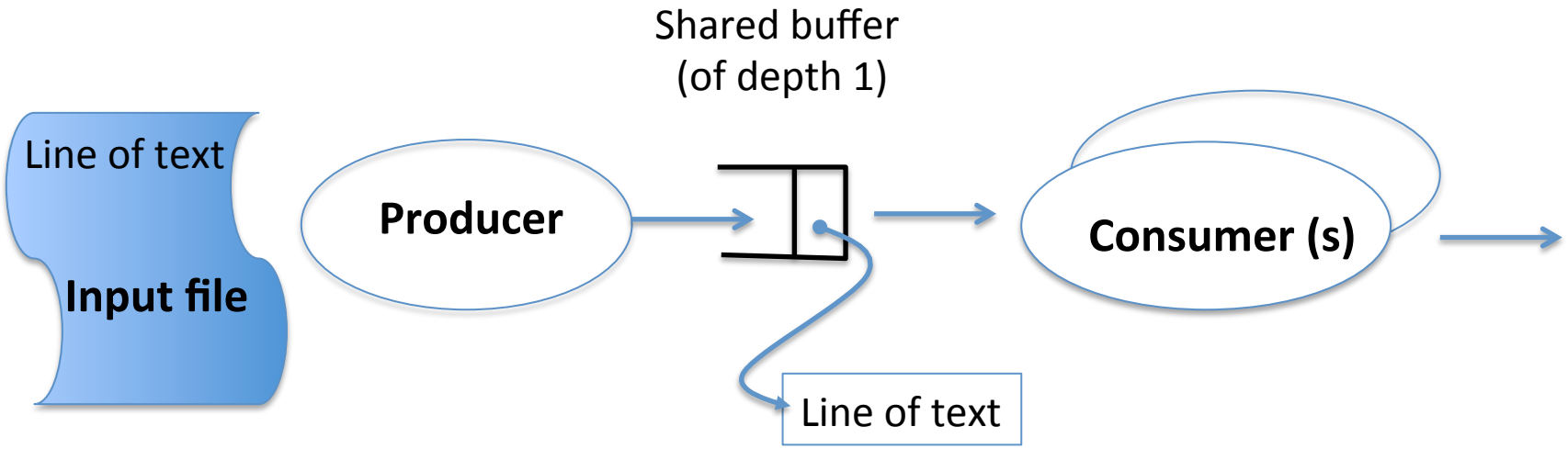
Objectives

- Demonstrate a structured way to approach concurrent programming (of threads)
  - Synchronized shared objects (in C!)
- Introduce the challenge of concurrent programming
- Develop understanding of a family of mechanisms
  - Flags, Locks, Condition Variables

Threads – the Faustian bargain

- Collections of cooperating sequential threads
  - Interact through shared variable
- Natural generalization of multiple (virtual) processors
- Performance
  - Overlap computation, I/O, and other compute
- Expressiveness
  - Progress on several fronts at once
- BUT …
  - Behavior depends on interleaving
  - Must be "correct" under all possible interleavings

Running Example



- Simplification of many typical use cases
- Producer can only fill the buffer if it is empty
- Consumers can only remove something from the buffer if it is full
- Doing so should empty it

A pthreads first cut (procon1.c)

```
int main (int argc, char *argv[])
{
  pthread_t prod;
  pthread_t cons;
  int rc;   long t;
  int *ret;
  FILE *rfile;
  so_t *share = malloc(sizeof(so_t));

  rfile = fopen((char *) argv[1], "r");

  share->rfile = rfile;
  share->line = NULL;
  pthread_create(&prod, NULL, producer, share);
  pthread_create(&cons, NULL, consumer, share);
  printf("main continuing\n");

  rc = pthread_join(prod, (void **) &ret);
  printf("main: producer joined with %d\n", *ret);
  rc = pthread_join(cons, (void **) &ret);
  printf("main: consumer joined with %d\n", *ret);
  pthread_exit(NULL);
  exit(0);
}
```

Producer -> Shared Object -> Consumer

```
void *producer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  FILE *rfile = so->rfile;
  int i;
  char *line;
  for (i = 0; (line = readline(rfile)); i++) {
    so->linenum = i;
    so->line = line;      /* share the line */
    fprintf(stdout, "Prod: [%d] %s", i, line);
  }
  printf("Prod: %d lines\n", i);
  *ret = i;
  pthread_exit(ret);
}
```

```
typedef struct sharedobject {
  FILE *rfile;
  int linenum;
  char *line;
} so_t;
```

```
void *consumer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  int i = 0;
  int len;
  char *line;
  while ((line = so->line)) {
    len = strlen(line);
    printf("Cons: [%d:%d] %s", i,
                so->linenum, line);
  }
  printf("Cons: %d lines\n", i);
  *ret = i;
  pthread_exit(ret);
}
```

Key Concepts

- **Race condition:** output of a concurrent program depends on the order of operations between threads
- **Atomic operations**: indivisible operations that cannot be interleaved with or split by other operations
- **Correctness (or safety):** "every line is processed by the consumer(s) exactly once".
  - under any possible scheduling
- Liveness: eventually every line gets produced and consumed
  - Neither waits indefinitely (under any possible scheduling)

NON-fixes: yield (procon2)

```
void *producer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  FILE *rfile = so->rfile;
  int i;
  char *line;
  for (i = 0; (line = readline(rfile)); i++) {
    so->linenum = i;
    so->line = line;      /* share the line */
    fprintf(stdout, "Prod: [%d] %s", i, line);
  }
  printf("Prod: %d lines\n", i);
  *ret = i;
  pthread_exit(ret);
}
```

pthread_yield

```
void *consumer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  int i = 0;
  int len;
  char *line;
  while ((line = so->line)) {
    len = strlen(line);
    printf("Cons: [%d:%d] %s", i,
                so->linenum, line);
  }
  printf("Cons: %d lines\n", i);
  *ret = i;
  pthread_exit(ret);
}
```

```
typedef struct sharedobject {
  FILE *rfile;
  int linenum;
  char *line;
} so_t;
```

NON-fixes: busywait (procon3-4)

```
void *producer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  FILE *rfile = so->rfile;
  int i;
  char *line;
  for (i = 0; (line = readline(rfile)); i++) {
    so->linenum = i;
    so->line = line;      /* share the line */
    fprintf(stdout, "Prod: [%d] %s", i, line);
  }
  printf("Prod: %d lines\n", i);
  *ret = i;
  pthread_exit(ret);
}
```

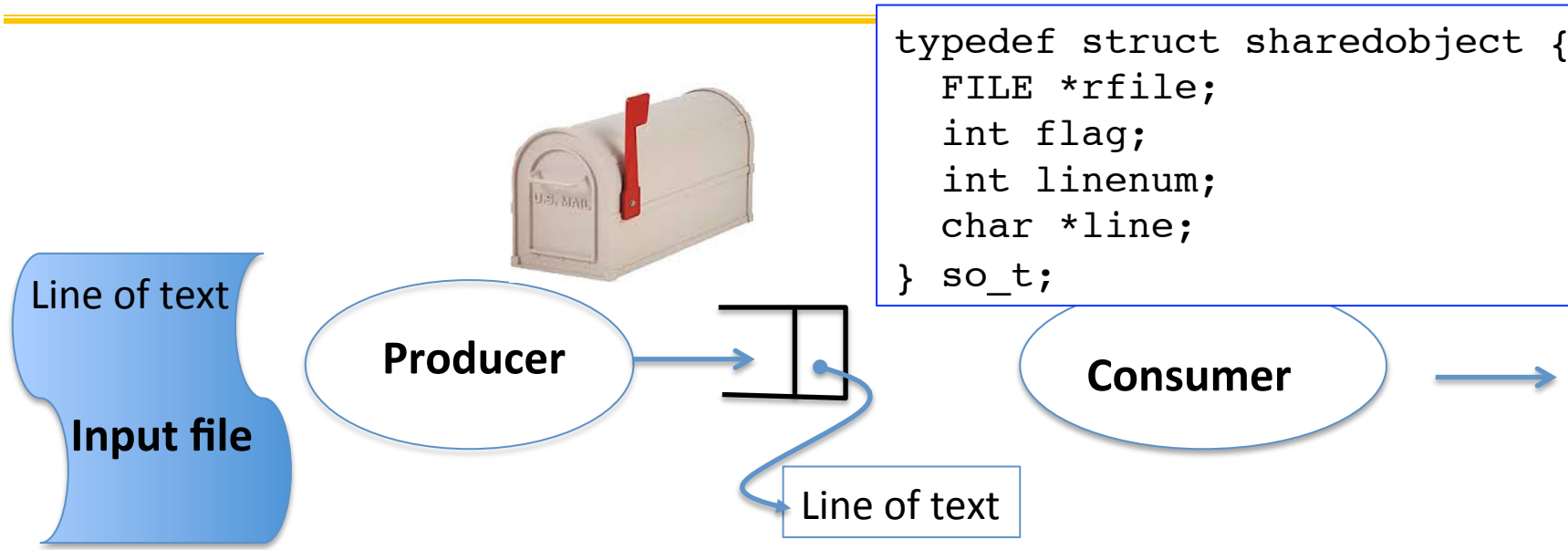
```
while (so->line) {
    printf("Prod wait %d\n",w++);
```

```
while (so->line == NULL)
  pthread_yield();
```

```
typedef struct sharedobject {
  FILE *rfile;
  int linenum;
  char *line;
} so_t;
```

```
void *consumer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  int i = 0;
  int len;
  char *line;
  while ((line = so->line)) {
    len = strlen(line);
    printf("Cons: [%d:%d] %s", i,
                so->linenum, line);
  }
  printf("Cons: %d lines\n", i);
  *ret = i;
  pthread_exit(ret);
}
```

Simplest synchronization: a flag



- Alternating protocol of a single producer and a single consumer can be coordinated by a simple flag
- Integrated with the shared object

```
int markfull(so_t *so) {
  so->flag = 1;
  while (so->flag) {}
  return 1;
}
```

```
int markempty(so_t *so) {
  so->flag = 0;
  while (!so->flag) {}
  return 1;
}
```

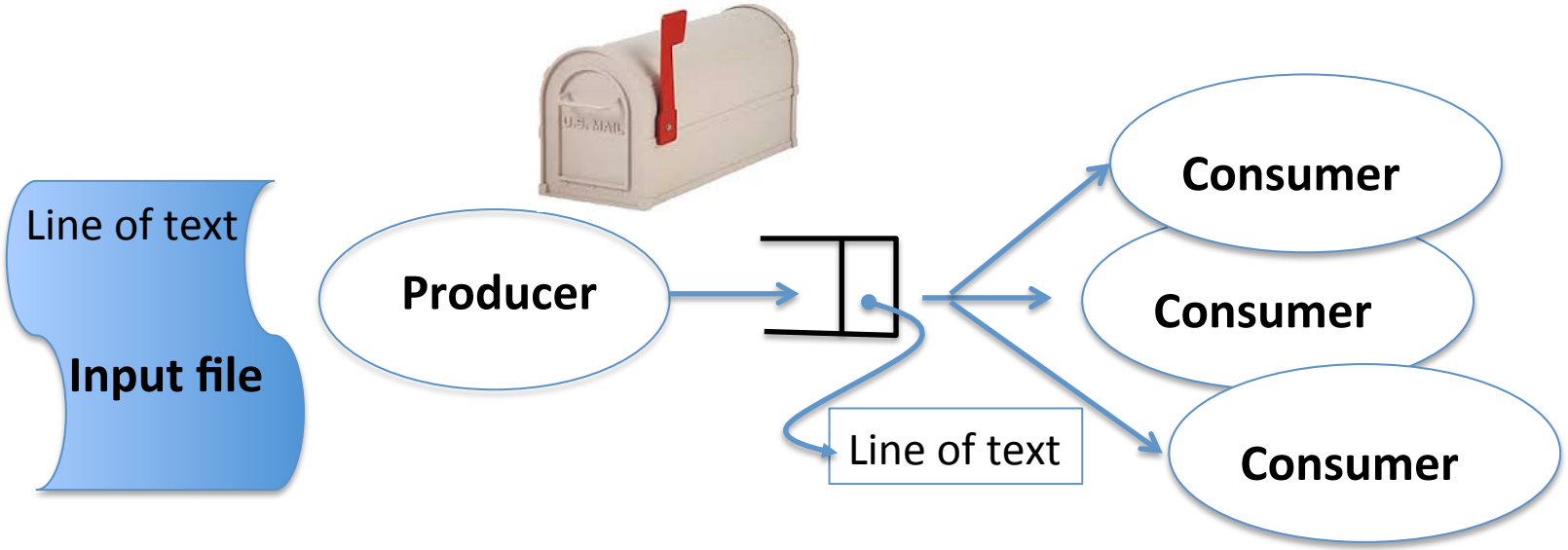
Almost fix: flags (proconflag.c)

```
void *producer(void *arg) {
 …
   for (i = 0; (line = readline(rfile)); i++) {
    so->linenum = i;
    so->line = line;
    markfull(so);
    fprintf(stdout, "Prod: [%d] %s",
                    i, line);
  }
  so->line = NULL;
  so->flag = 1;
  printf("Prod: %d lines\n"
  *ret = i;
  pthread_exit(ret);
}
```

Must preserve write ordering !!!

```
void *consumer(void *arg) {
 …
  while (!so->flag) {} /* wait for proc
  while ((line = so->line)) {
    i++;
    len = strlen(line);
    printf("Cons: [%d:%d] %s", i,
                    so->linenum, line);
    markempty(so);
  }
  so->flag = 0;
  printf("Cons: %d lines\n", i);
  *ret = i;
  pthread_exit(ret);
```

Multiple Consumers, etc.



- More general relationships require mutual exclusion
  - Each line is consumed exactly once!

Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock: prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- unlock when leaving, after done accessing shared data
- wait if locked (all synch involves waiting!)

Fork-Join Model (proNcon2)

```
int main (int argc, char *argv[])
{
  pthread_t prod;
  pthread_t cons[CONSUMERS];
  targ_t carg[CONSUMERS];
  …
  so_t *share = malloc(sizeof(so_t));

  share->rfile = rfile;
  share->line = NULL;
  share->flag = 0;

  pthread_create(&prod, NULL, producer, share);
  for (i=0; i<CONSUMERS; i++) {
    carg[i].tid = i;
    carg[i].soptr = share;
    pthread_create(&cons[i], NULL, consumer, &carg[i]);
  }

  rc = pthread_join(prod, (void **) &ret);
  for (i=0; i<CONSUMERS; i++)
    rc = pthread_join(cons[i], (void **) &ret);
  pthread_exit(NULL);
  exit(0);
}
```

Incorporate Mutex into shared object

- Methods on the object provide the synchronization
  - Exactly one consumer will process the line

```
typedef struct sharedobject {
  FILE *rfile;
  pthread_mutex_t solock;
  int flag;
  int linenum;
  char *line;
} so_t;
```

```
int waittill(so_t *so, int val) {
    while (1) {
    pthread_mutex_lock(&so->solock);
    if (so->flag == val)
        return 1; /* rtn with object locked */
    pthread_mutex_unlock(&so->solock);
  }
}
int release(so_t *so) {
  return pthread_mutex_unlock(&so->solock);
}
```

Single Consumer – Multi Consumer

```
void *producer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  FILE *rfile = so->rfile;
  int i;
  int w = 0;
  char *line;
  for (i = 0; (line = readline(rfile)); i++) {
    waittill(so, 0);              /* grab lock when empty */
    so->linenum = i;              /* update the shared state */
    so->line = line;              /* share the line */
    so->flag = 1;                 /* mark full */
    release(so);                  /* release the loc */
    fprintf(stdout, "Prod: [%d] %s", i, line);
  }
  waittill(so, 0);                /* grab lock when empty */
  so->line = NULL;
  so->flag = 1;
  printf("Prod: %d lines\n", i);
  release(so);            /* release the loc */
  *ret = i;
  pthread_exit(ret);
}
```

Continued (proNcon3.c)

```
void *consumer(void *arg) {
  targ_t *targ = (targ_t *) arg;
  long tid = targ->tid;
  so_t *so = targ->soptr;
  int *ret = malloc(sizeof(int));
  int i = 0;;
  int len;
  char *line;
  int w = 0;
  printf("Con %ld starting\n",tid);
  while (waittill(so, 1) &&
         (line = so->line)) {
    len = strlen(line);
    printf("Cons %ld: [%d:%d] %s", tid, i, so->linenum, line);
    so->flag = 0;
    release(so);                /* release the loc */
    i++;
  }
  printf("Cons %ld: %d lines\n", tid, i);
  release(so);           /* release the loc */
  *ret = i;
  pthread_exit(ret);
}
```

Initialization

```
share->line = NULL;
share->flag = 0;                    /* initially empty */
pthread_mutex_init(&share->solock, NULL);

pthread_create(&prod, NULL, producer, share);

for (i=0; i<CONSUMERS; i++) {
  carg[i].tid = i;
  carg[i].soptr = share;
  pthread_create(&cons[i], NULL, consumer, &carg[i]);
}
printf("main continuing\n");

rc = pthread_join(prod, (void **) &ret);
printf("main: producer joined with %d\n", *ret);
for (i=0; i<CONSUMERS; i++) {
  rc = pthread_join(cons[i], (void **) &ret);
  printf("main: consumer %d joined with %d\n", i, *ret);
}
share->flag = 0;
pthread_mutex_destroy(&share->solock);
pthread_exit(NULL);
```

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Eliminate the busy-wait?

- Especially painful since looping on lock/unlock of highly contended resource

```
typedef struct sharedobject {
  FILE *rfile;
  pthread_mutex_t solock;
  int flag;
  int linenum;
  char *line;
} so_t;
```

```
int waittill(so_t *so, int val) {
    while (1) {
    pthread_mutex_lock(&so->solock);
    if (so->flag == val)
        return 1; /* rtn with object locked */
    pthread_mutex_unlock(&so->solock);
  }
}
int release(so_t *so) {
  return pthread_mutex_unlock(&so->solock);
}
```

Condition Variables

- Wait: atomically release lock and relinquish processor until signalled
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

- Called only when holding a lock !!!!

In the object

```
typedef struct sharedobject {
  FILE *rfile;
  pthread_mutex_t solock;
  pthread_cond_t flag_cv;
  int flag;
  int linenum;
  char *line;
} so_t;
```

```
int waittill(so_t *so, int val, int tid) {
  pthread_mutex_lock(&so->solock);
  while (so->flag != val)
    pthread_cond_wait(&so->flag_cv, &so->solock);
return 1;
}

int release(so_t *so, int val, int tid) {
  so->flag = val;
  pthread_cond_signal(&so->flag_cv);
  return pthread_mutex_unlock(&so->solock);
}
int release_exit(so_t *so, int tid) {
  pthread_cond_signal(&so->flag_cv);
  return pthread_mutex_unlock(&so->solock);
}
```

Critical Section

```
void *producer(void *arg) {
  so_t *so = arg;
  int *ret = malloc(sizeof(int));
  FILE *rfile = so->rfile;
  int i;
  int w = 0;
  char *line;
  for (i = 0; (line = readline(rfile)); i++) {
    waittill(so, 0, 0);          /* grab lock when empty */
    so->linenum = i;             /* update the shared state */
    so->line = line;             /* share the line */
    release(so, 1, 0);           /* release the loc */
    fprintf(stdout, "Prod: [%d] %s", i, line);
  }
  waittill(so, 0, 0);            /* grab lock when empty */
  so->line = NULL;
  release(so, 1, 0);             /* release it full and NULL */
  printf("Prod: %d lines\n", i);
  *ret = i;
  pthread_exit(ret);
}
```

Change in invariant on exit

```
void *consumer(void *arg) {
  targ_t *targ = (targ_t *) arg;
  long tid = targ->tid;
  so_t *so = targ->soptr;
  int *ret = malloc(sizeof(int));
  int i = 0;;
  int len;
  char *line;
  int w = 0;
  printf("Con %ld starting\n",tid);
  while (waittill(so, 1, tid) &&
         (line = so->line)) {
    len = strlen(line);
    printf("Cons %ld: [%d:%d] %s", tid, i, so->linenum, line);
    release(so, 0, tid);                    /* release the loc */
    i++;
  }
  printf("Cons %ld: %d lines\n", tid, i);
  release_exit(so, tid);                    /* release the loc */
  *ret = i;
  pthread_exit(ret);
}
```

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock
  - What if wait, then release? What if release, then wait?

```
int waittill(so_t *so, int val, int tid) {
  pthread_mutex_lock(&so->solock);
  while (so->flag != val)
    pthread_cond_wait(&so->flag_cv, &so->solock);
  return 1;
}
```

Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while (needToWait())
    condition.Wait(lock);
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
  - In Pintos kernel, everything!
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) condition.Wait(lock);
  - Do not assume when you wake up, signaller just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

Mesa vs. Hoare semantics

- Mesa (in textbook, Hansen)
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller
  - Nested signals possible!

Implementing Synchronization

Concurrent Applications

Semaphores | Locks | Condition Variables

Interrupt Disable | Atomic Read/Modify/Write Instructions

Multiple Processors | Hardware Interrupts