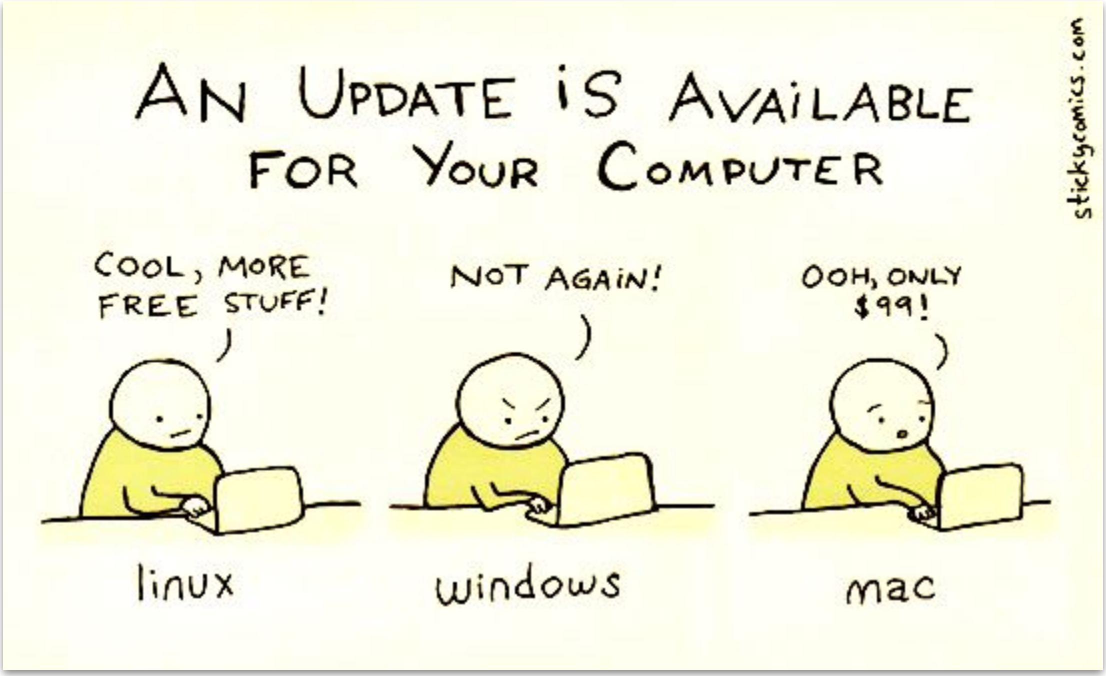


AN UPDATE IS AVAILABLE FOR YOUR COMPUTER
stickycomics.com
COOL, MORE FREE STUFF!
NOT AGAIN!
OOH, ONLY $99!
linux
windows
mac

CPSC 457

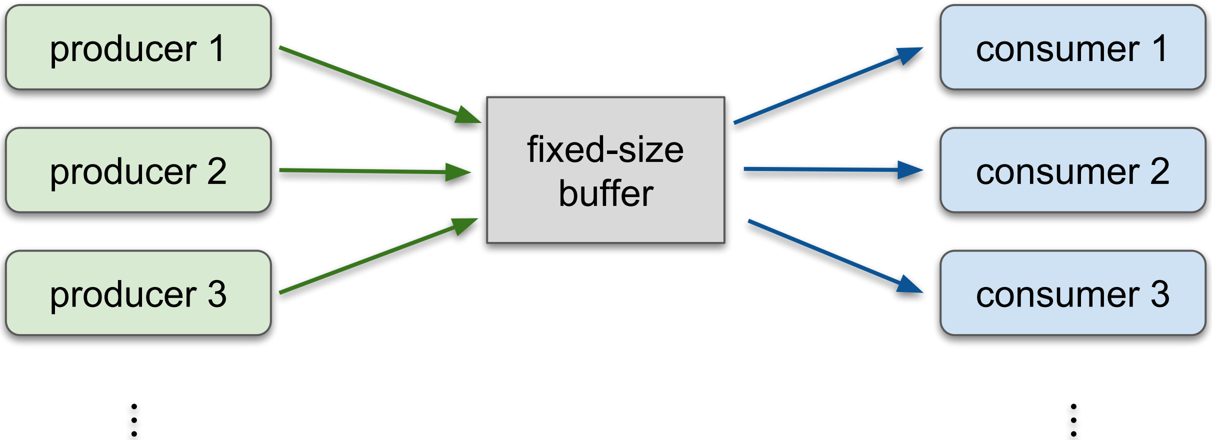
Producer/consumer problem
Condition variables and Semaphores

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Outline

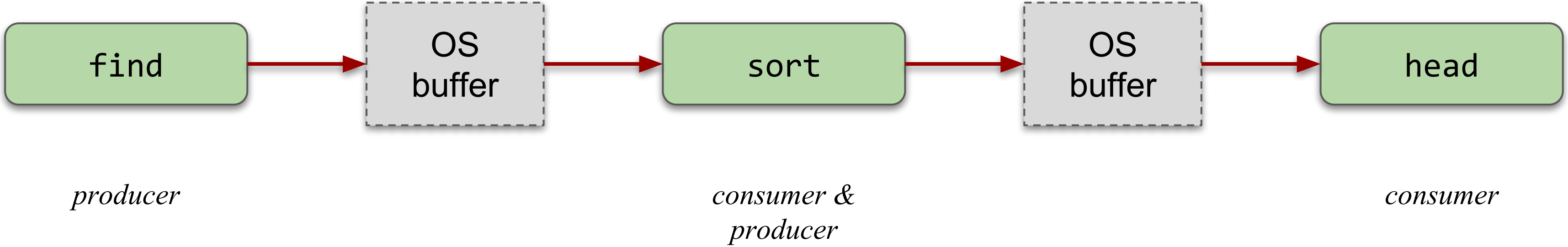
- producer-consumer problem
- mutexes and condition variables
- semaphores

Producer-consumer problem



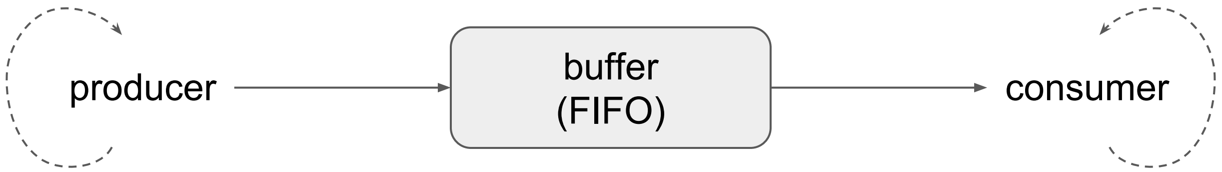
Pipes...

```
$ find . -type f -printf "%-20s%p\n" | sort -nr | head -n 10
```



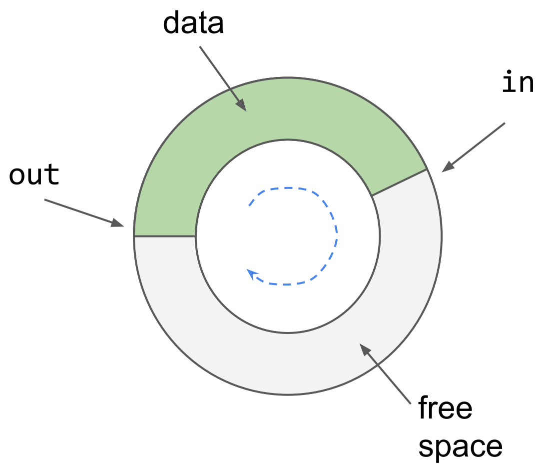
The producer-consumer problem

- simplest case: one consumer and one producer processes/threads
- the two processes or threads share a **fixed-size buffer**, used as a queue
- producer puts data into buffer, must wait if buffer full
- consumer takes data out of the buffer, must wait if buffer empty

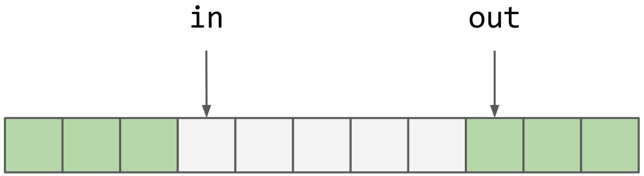


Circular buffer

Concept:



Implementation:



- common way to implement fixed-size queue

```
typedef struct { … } item;
item buffer[BUFF_SIZE];
int in = 0; // next free position
int out = 0; // next filled position
int count = 0; // number of items in buffer
```

- buffer is empty when:

  `in == out` or when `count == 0`

- buffer is full when:

  `(in + 1) % BUFF_SIZE == out`

  or when `count == BUFF_SIZE`

Possible implementation - naive

Thread1 — producer thread

```
while(1) {
  item = produceItem();
  // wait while buffer is full
  while((in+1) % BUFF_SIZE == out){}
  // insert item into buffer
  buffer[in] = item;
  in = (in + 1) % BUFF_SIZE;
  count ++;
}
```

Thread 2 — consumer thread

```
while(1) {
  // wait while buffer is empty
  while( in == out){}
  // remove item from buffer
  item = buffer[out];
  out = (out+1) % BUFF_SIZE;
  count --;
  consumeItem(item);
}
```

- there is a race condition if we run the above in two different threads
- where?

Possible implementation - naive

Thread1 — p

```
reg1   = count
reg1   = reg1 + 1
count = reg1
```

```
while(1) {
  item = produceIte
  // wait while bu     is full
  while((in+1) %     F_SIZE == out){}
  // insert ite    into buffer
  buffer[in]     item;
  in = (in     1) % BUFF_SIZE;
  count ++;
}
```

Thread

```
reg2   = count
reg2   = reg2 - 1
count = reg2
```

```
while(
  // wait whil         er is empty
  while( in =          ){}
  // remove       m from buffer
  item = bu    fer[out];
  out = (  ut+1) % BUFF_SIZE;
  count --;
  consumeItem(item);
}
```

Possible execution sequence (starting eg. with count=5):

```
T1: reg1   = count          // count=5
T2: reg2   = count          // count=5
T2: reg2   = reg2 - 1       // count=5
T2: count = reg2            // count=4
```

```
T1: reg1   = reg1 + 1       // count=4
T1: count = reg1            // count=6
```

Possible implementation with a mutex

Thread1 — producer thread

```
while(1) {
  item = produceItem();
  // wait while buffer is full
  while((in+1) % BUFF_SIZE == out){}
  // insert item into buffer
  buffer[in] = item;
  in = (in + 1) % BUFF_SIZE;
  pthread_mutex_lock(& mut);
    count ++; // critical section
  pthread_mutex_unlock(& mut);
}
```

Thread 2 — consumer thread

```
while(1) {
  // wait while buffer is empty
  while( in == out){}
  // remove item from buffer
  item = buffer[out];
  out = (out+1) % BUFF_SIZE;
  pthread_mutex_lock(& mut);
    count --; // critical section
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

- Let's protect the counter using a mutex to remove the race condition with count.
- Can you spot any other problems?

Possible implementation with a mutex

Thread1 — producer thread

```
while(1) {
  item = produceItem();
  // wait while buffer is full
  while((in+1) % BUFF_SIZE == out){}
  // insert item into buffer
  buffer[in] = item;
  in = (in + 1) % BUFF_SIZE;
  pthread_mutex_lock(& mut);
    count ++;
  pthread_mutex_unlock(& mut);
}
```

Thread 2 — consumer thread

```
while(1) {
  // wait while buffer is empty
  while( in == out){}
  // remove item from buffer
  item = buffer[out];
  out = (out+1) % BUFF_SIZE;
  pthread_mutex_lock(& mut);
    count --;
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

- this solution would only work for one producer thread and one consumer thread
- what if we wanted multiple producers and/or multiple consumers?
- another important problem: busy wait!

Possible implementation with a mutex (v2)

Thread1 — producer thread

```
while(1) {
  item = produceItem();
  pthread_mutex_lock(& mut);
    // wait while buffer is full
    while((in+1) % BUFF_SIZE == out){}
    // insert item into buffer
    buffer[in] = item;
    in = (in + 1) % BUFF_SIZE;
    count ++;
  pthread_mutex_unlock(& mut);
}
```

Thread 2 — consumer thread

```
while(1) {
  pthread_mutex_lock(& mut);
    // wait while buffer is empty
    while( in == out){}
    // remove item from buffer
    item = buffer[out];
    out = (out+1) % BUFF_SIZE;
    count --;
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

- let's make one big critical section
- this naive attempt to add support for multiple consumers/producers unfortunately leads to a **deadlock**
- can you find it?

Possible implementation with a mutex (v2)

Thread1 — producer thread

```
while(1) {
  item = produceItem();
  pthread_mutex_lock(& mut);
    // wait while buffer is full
    while((in+1) % BUFF_SIZE == out){}
    // insert item into buffer
    buffer[in] = item;
    in = (in + 1) % BUFF_SIZE;
    count ++;
  pthread_mutex_unlock(& mut);
}
```

2

Thread 2 — consumer thread

```
while(1) {
  pthread_mutex_lock(& mut);
    // wait while buffer is empty
    while( in == out){}
    // remove item from buffer
    item = buffer[out];
    out = (out+1) % BUFF_SIZE;
    count --;
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

1

- one possible deadlock: buffer is empty, and consumer enters its critical section ...
  - consumer spins inside CS, producer blocks on trying to acquire mutex
- another deadlock: can you find it?

Condition variables (CVs)

- condition variables are another type of synchronization primitives
- used together with mutexes
- perfect for implementing critical sections containing loops waiting for some 'condition'

```
// critical section protected with mutex m
lock(m)
  …
  while( ! some_condition ) {;}
  …
unlock(m)
```

- where the condition can only become true if another thread runs it's critical section
- CVs make it simple to implement critical sections like the one above

Condition variables

a common pattern of using CVs:

- a thread enters it's critical section (locks a mutex)
- inside CS, thread needs to wait for some condition to become true
- but the condition can only become true by allowing some other thread to lock the mutex
- to facilitate this, the thread puts itself to sleep and simultaneously releases the mutex

```
// critical section protected with mutex m
lock(m)
  …
  while( ! some_condition ) { wait(cv); }
  …
unlock(m)
```

- now some other thread can lock the mutex and execute code that will satisfy the condition

Condition variables

- eventually some other thread
  - locks the mutex (optional)
  - changes some state that will satisfy the condition
  - notifies the waiting thread via the condition variable
  - releases mutex (optional)

```
// some other thread...
lock(m)
  …
  some_condition = TRUE;
  signal(cv);
  …
unlock(m)
```

- the waiting thread then wakes up, and acquires the mutex back automatically

Condition variables

```
pthread_mutex_t mutex; // mutex
pthread_cond_t cond;   // condition variable
```

- **`pthread_cond_wait(&cond, &mutex);`**
  - atomically releases mutex and causes the calling thread to block, until some other thread calls `pthread_cond_signal(&cond)`
  - after returning, the mutex is automatically re-acquired
  - after returning, the condition must be rechecked !!! (spurious wakeups)
- **`pthread_cond_signal(&cond);`**
  - wakes up one thread waiting on `cond`
  - if no threads waiting on `cond`, the signal is lost
  - must be followed by `pthread_mutex_unlock()` if the blocked thread uses the same mutex

Condition variables

- `pthread_cond_init(& cond, & attr)`
  - creates condition variable
- `pthread_cond_destroy(& cond)`
  - destroys a condition variable
- `pthread_cond_broadcast(& cond)`
  - wakes up **all** threads waiting on the condition

Condition variable example

thread 1:

```
while(1) {
  pthread_mutex_lock(&mutex);
    while(count == 0) {
      pthread_cond_wait(&cond, &mutex);
    }
    counter --;
  pthread_mutex_unlock(&mutex);
}
```

thread 2:

```
while(1) {
  pthread_mutex_lock(&mutex);
    counter ++;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

- thread 1 - decrementing counter, but never below 0
- thread 2 - incrementing counter
- no deadlocks
- no busy waiting

Let's fix this

- deadlock due to one thread stuck in an infinite loop after locking mutex
- while other thread has no chance to run its CS to allow the other thread to exit the loop

Thread1 – producer thread

```
while(1) {
  item = produceItem();
  pthread_mutex_lock(& mut);
    // wait while buffer is full
    while((in+1) % BUFF_SIZE == out){}
    // insert item into buffer
    buffer[in] = item;
    in = (in + 1) % BUFF_SIZE;
    count ++;
  pthread_mutex_unlock(& mut);
}
```

1

Thread 2 – consumer thread

2

```
while(1) {
  pthread_mutex_lock(& mut);
    // wait while buffer is empty
    while( in == out){}
    // remove item from buffer
    item = buffer[out];
    out = (out+1) % BUFF_SIZE;
    count --;
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

Consumer/producer with condition variables

```
pthread_mutex_t mut;
pthread_cond_t full, empty;
```

producer thread:

```
while(1) {
  item = produceItem();
  pthread_mutex_lock(& mut);
    while((in+1) % BUFF_SIZE == out){
      pthread_cond_wait(& full,& mut);
    }
    buffer[in] = item;
    in = (in + 1) % BUFF_SIZE;
    count ++;
  pthread_mutex_signal(& empty);
  pthread_mutex_unlock(& mut);
}
```

consumer thread:

```
while(1) {
  pthread_mutex_lock(& mut);
    while( in == out){
      pthread_cond_wait(& empty,& mut);
    }
    item = buffer[out];
    out = (out+1) % BUFF_SIZE;
    count --;
  pthread_mutex_signal(& full);
  pthread_mutex_unlock(& mut);
  consumeItem(item);
}
```

Semaphore

- another synchronization primitive
- think of semaphore as a special integer variable used for signalling among processes
- the value could indicate number of available units of some resource
- supports three operations:
- **initialization**
  - can be initialized with any value (0 … max)
- **decrement**
  - reduce semaphore by 1,
  - blocks the calling process if going below 0

    `down(s)`, `wait(S)` or `sem_wait(S)`
- **increment**
  - increase value by 1;
  - and possibly unblock another blocked process

    `up(S)`, `signal(S)` or `sem_post(S)`

can be used to protect critical sections similar to a mutex

```
semaphore s;

wait(s);
  // Critical Section
signal(s);
```

Semaphore

- can be used to protect critical sections, similar to a mutex

```
semaphore s;

wait(s);
  // Critical Section
signal(s);
```

- each semaphore maintains a queue of processes blocked on the semaphore
- when a semaphore is locked by a thread, it can be unlocked by **any** thread
- as opposed to mutex, where a locking/unlocking must be done by the same thread

Binary semaphore

- special type of semaphore with value either 0 or 1
- possible pseudocode implementation:

`void wait(int s) {`
`    while (s == 0) {;}`
`    s = 0;`
`}`
	`void signal(s) {`
	`        s = 1;`
	`}`

  where the bodies are executed atomically
- atomic operation:
  - is an operation that appears to execute instantaneously with respect to the rest of the system
  - eg. cannot be interrupted by signals, threads, interrupts, ...

Counting semaphore

- aka. **general semaphore**, or just **semaphore**
- represents an integer value S
- S > 0: value of S is the number of processes/threads that can issue a wait and immediately continue to execute
- S = 0: eg. all resources are busy, the calling process/thread must wait
- S < 0: some implementations might do this, `abs(S)` then represents the number of processes that are waiting to be unblocked
- possible pseudocode implementation:

```
void wait(int s) {
    while (s <= 0) {;}
    s --;
}
```

```
void signal(s) {
      s ++;
}
```

where the bodies are executed atomically

Counting semaphore

- aka. **general semaphore**, or just **semaphore**
- represents an integer value S

`S > 0`	value of S is the number of threads that can issue `wait()` without blocking
`S == 0`	all resources are busy, any thread that calls `wait()` will block
`S < 0`	some implementations allow this… `|S|` then represents the number of threads that are blocked on `wait()`

- possible pseudocode implementation:

```
void wait(int s) {
    while (s <= 0) {;}
    s --;
}
```

```
void signal(s) {
     s ++;
}
```

where the bodies are executed atomically

Semaphores vs. condition variables

- `signal()` compared to `cv_signal()`:
  - `cv_signal()` is lost (has no effect) if no thread is waiting
  - `signal()` increments the semaphore always, and possibly wakes up a thread
- `wait()` compared to `cv_wait()`:
  - `cv_wait()` does not check the condition, it always blocks
  - `wait()` checks the value of the semaphore, and may or may not block

POSIX semaphores

- `int sem_init (sem_t *sem, int pshared, unsigned int value)`
  initializes semaphore to `value`, `pshared=0/1` shared by threads/processes
- `int sem_destroy (sem_t * sem)`
  destroys the semaphore, fails if some threads are waiting on it
- `int sem_wait (sem_t * sem)`
  suspends the calling thread until the semaphore is non-zero, then atomically decreases the semaphore count
- `int sem_post (sem_t * sem)`
  atomically increases the semaphore, never blocks, may unblock blocked threads, safe to use in signal handlers in Linux on 486+ hardware
- `int sem_getvalue (sem_t * sem, int * sval)`
  returns the value of semaphore via `sval`
- `int sem_trywait (sem_t * sem)`
  non blocking version of `sem_wait()`

Example with semaphore = 2

```
sem_t s;
…
sem_init( & s, 0, 2); // initialize semaphore to 2
```

```
Thread 1:

while(1) {
  wait (s);
  // critical section
  signal (s);
}
```

```
Thread 2:

while(1) {
  wait (s);
  // critical section
  signal (s);
}
```

```
Thread 3:

while(1) {
  wait (s);
  // critical section
  signal (s);
}
```

- suppose 3 threads try to enter their CSs protected by a semaphore

Example with semaphore = 2

```
sem_t s;
...
sem_init( & s, 0, 2); // initialize semaphore to 2
```

```
Thread 1:

while(1) {
  wait (s);
  // critical section
  signal (s);
}
```

```
Thread 2:

while(1) {
  wait (s);
  // critical section
  signal (s);
}
```

```
Thread 3:

while(1) {
  wait (s);
  // critical section
  signal (s);
}
```

- two threads will enter their CS simultaneously
- the other thread will be blocked

Example with semaphore = 2

```
sem_t s;
...
sem_init( & s, 0, 2); // initialize semaphore to 2
```

```
Thread 1:

while(1) {
  wait (s);
  // critical section
  signal (s);
}
```

```
Thread 2:

while(1) {
  wait (s);
  // critical section
  signal (s);
}
```

```
Thread 3:

while(1) {
  wait (s);
  // critical section
  signal (s);
}
```

- as soon as one thread leaves CS, the last thread will be allowed to enter its CS

Improved semaphore implementation, with a queue & blocking

- `S->list` is a list of processes/threads
- a process can `block()` itself
- a process can be unblocked by `wakeup()`
- `getpid()` returns process or thread ID

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

```
void wait(semaphore *S) {
    S->value --;
    if (S->value < 0) {
        S->list->push(getpid());
        block();
    }
}
```

```
void signal(semaphore *S) {
    S->value ++;
    if (S->value <= 0) {
        pid = S->list->pop();
        wakeup(pid);
    }
}
```

It is tricky

- one subtle error and everything comes to a grinding halt
- concurrent programming can be more difficult than coding in assembly
- any error with semaphores will potentially result in race conditions, deadlocks, and other forms of unpredictable and irreproducible behaviour
- for example:

Violate mutual exclusivity:

```
signal(sem);
  ...
  critical section
  ...
wait(sem);
```

Deadlock:

```
lock(mutex);
  ...
  critical section
  ...
lock(mutex);
```

Another common problem with semaphores

- Scenario: managing a pool of N resources
  - semaphore S is used to keep track of the # of available resources
  - initialization: `sem_init(S,0,N)`
  - each process may need K_i <= N resources at a time
  - you might be tempted to accomplish this by K_i consecutive invocations of wait(S)

- Can you see the problem?
  - Example: N = 10, thread 1 needs 7 resources and thread 2 needs 6 resources
  - depending on scheduling, we may get a **deadlock**

Another common problem with semaphores


```
Thread 1:

for(i=0; i<7; i++)
  wait(S);

// critical section

for(i=0; i<7; i++)
  signal(S);
...
```

```
Thread 2:

for(i=0; i<6; i++)
  wait(S);

// critical section

for(i=0; i<6; i++)
  signal(S);
...
```

- order of operations
- thread 1 requests 6 resources, then scheduler switches to thread 2
- thread 2 requests 4 resources, exhausting all available resources
- both threads are stuck → **deadlock**

Extended semaphore - possible implementation


```
void wait(semaphore *S, int k) {
     S->value -= k;
     if (S->value < 0) {
        S->list->push(getpid());
        block();
}
```

```
void signal(semaphore *S, int k) {
    S->value += k;
    if (S->value >= 0) {
        pid = S->list->pop();
        wakeup(pid);
}
```

Thread 1:	Thread 2:

`wait(S, 7);`
`// critical section`
`signal(S, 7);`
	`wait(S, 6);`
	`// critical section`
	`signal(S, 6);`

Summary

- producer-consumer problem
- condition variables
- semaphores (binary, counting, extended)

Reference: 2.3 (Modern Operating Systems)

5.5, 5.8, 6.5 (Operating System Concepts)

Additional reading:

- The Little Book of Semaphores

  http://greenteapress.com/wp/semaphores/

Review

- Race conditions are not a problem among processes, only among threads. True or False?
- What is the main difference between `pthread_cond_signal()` for mutex and `sem_post()` for semaphore?
- A mutex is identical to binary semaphore. True or False?
- What does the value of the semaphore tell you?
- Define atomic operation.

Dining philosophers with semaphores


```
#define N 5 /* number of philosophers */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */

int state[N]; /* each philosophers is either THINKING, HUNGRY or EATING, initialized to THINKING */
sem_t cs_m; /* semaphore for the critical section, shared by all philosophers, initialized to 1 */
sem_t p_m[N]; /* one semaphore per philosopher, initialized to 0 */

int left(i) { return (i+N-1) % N; }
int right(i) { return (i+1) % N; }

void philosopher(int i) /* to be executed by different threads */
{
  while (1) {
    think( ); /* philosopher is thinking */
    take_forks(i); /* acquire two forks or block */
    eat( ); /* yum-yum, spaghetti */
    put_forks(i); /* put both forks back on table */
  }
}
```

Dining philosophers with semaphores

```
void take_forks(int i) {
  down(& cs_m);             /* enter critical region */
    state[i] = HUNGRY;      /* record fact that philosopher i is hungry */
    test(i);                /* try to acquire 2 forks */
  up(& cs_m);               /* exit critical region */
  down(& p_m[i]);           /* block if forks were not acquired */
}

void put_forks(i) {         /* i: philosopher number, from 0 to N-1 */
  down(& cs_m);             /* enter critical region */
    state[i] = THINKING;    /* philosopher has finished eating */
    test(left(i));          /* see if left neighbor can now eat */
    test(right(i));         /* see if right neighbor can now eat */
  up(& cs_m);               /* exit critical region */
}

void test(i) {              /* i: philosopher number, from 0 to N-1 */
  if (state[i] == HUNGRY && state[left(i)] != EATING && state[right(i)] != EATING) {
    state[i] = EATING;      /* only start eating if hungry, and neighbors not eating */
    up(& p_m[i]);
  }
}
```

Questions?