

Process Management II

Operating Systems

Oleg Goldshmidt

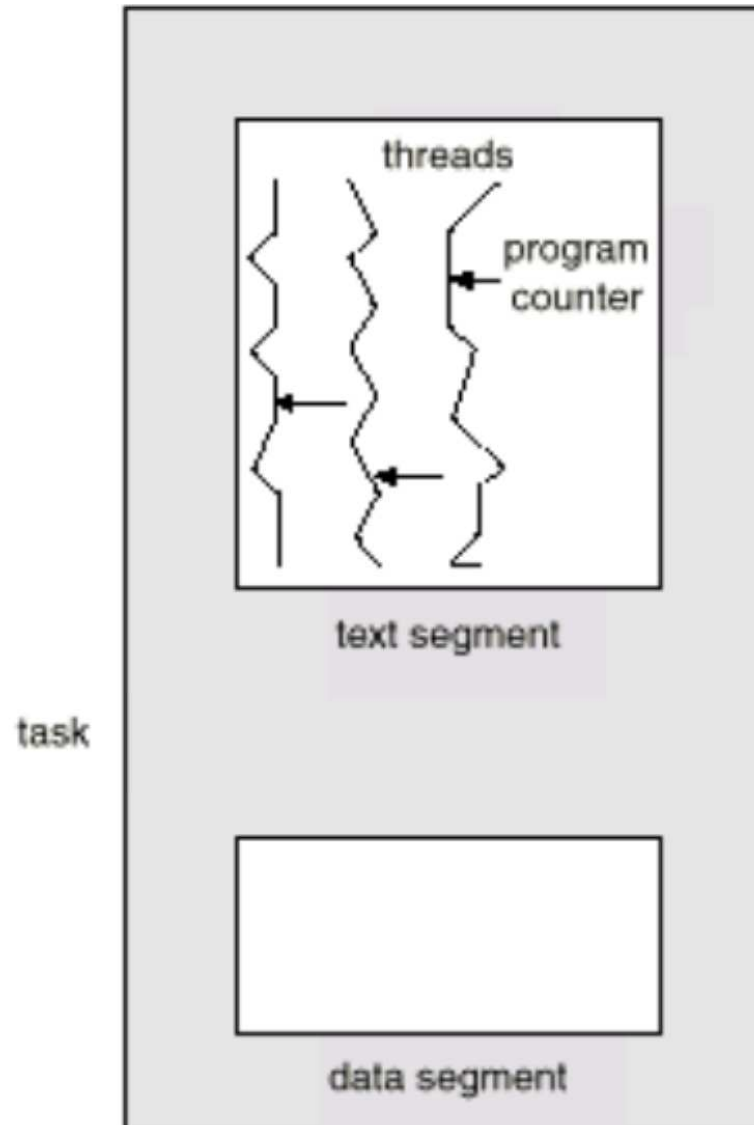
`ogoldshmidt@computer.org`

Lecture 3

Threads

- a **thread** (a.k.a. “lightweight process”) is a basic unit of CPU utilization
- a thread consists of
 - program counter
 - register set
 - stack
- a thread must belong to exactly one **task** (or process)
- threads in one task share all resources except CPU:
 - code section
 - data section
 - OS resources
- a traditional process is a task with one thread

Multiple Threads in a Process



Threads vs. Processes: Similarities

- both share the CPU
 - only one thread (of one process) may be running
- same (similar) states
 - ready, blocked, running, terminated
- can create children
- block on system calls
 - if one thread is blocked another can run

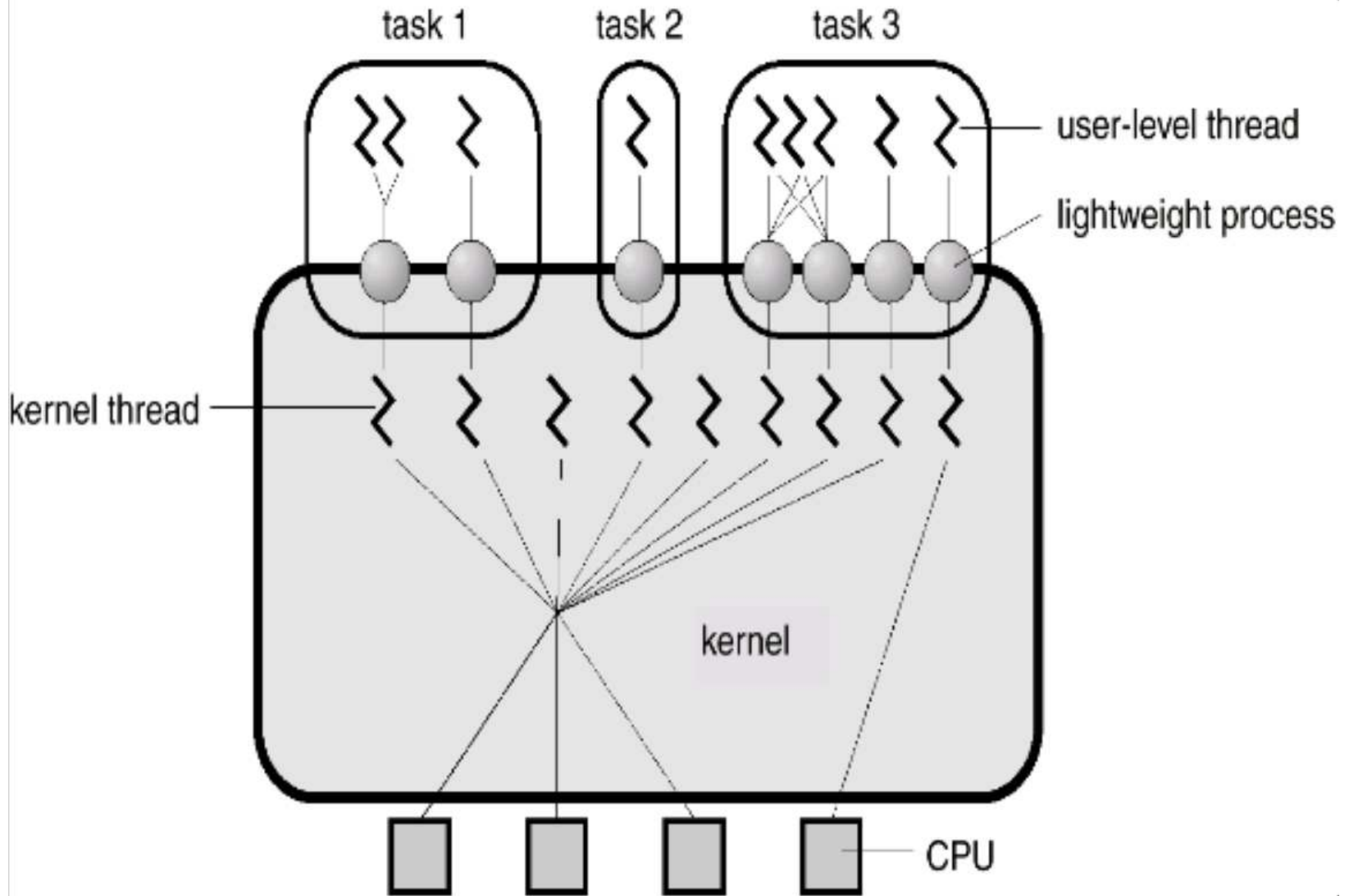
Threads vs. Processes: Differences

- threads have access to all the memory space of the task
 - can read from and write to each other's stacks
 - do not have — or require — mutual protection
- processes are useful to parallelize unrelated, independent tasks
- threads are useful to parallelize operations on shared resources
 - cooperation of multiple threads in the same task leads to higher throughput, improved performance
 - applications that require sharing common data (e.g., the buffer in producer-consumer) benefit from threads

Implementing Threads

- user-level threads
 - fast switching
 - write your own scheduler
 - but the OS can take the CPU away from the task
 - unfair scheduling since the OS allocates the CPU to tasks regardless of the number of threads
 - will not work with single-threaded kernels
 - imagine a thread executing a system call
- kernel supported threads
 - more expensive switching
 - fair scheduling is possible
 - a process can execute concurrent system calls
- hybrid (user-level and kernel-level) — Solaris 2

Solaris Threads



Cooperating Processes

- independent and cooperating processes
 - independent processes cannot affect each other's execution
 - cooperating processes can affect or be affected by the execution of another process
- what for?
 - information sharing
 - speedup
 - overlap CPU and I/O
 - use several processors
 - modularity
 - convenience

Cooperating Processes: Classic Models

- producer-consumer: one process produces information that is consumed by another
 - bounded-buffer — a buffer of fixed size is assumed
 - unbounded-buffer — no practical limit is placed on the buffer size
- reader-writer: a number of processes read from and write to a shared location in memory
- environment
 - shared memory
 - read and write atomicity
 - serial consistency: read returns the value stored by the latest write
 - message passing (distributed environments)

Shared Memory Bounded-Buffer

```
/* shared data */
item_t buffer[LENGTH];
/* in - next free position;
   out - first full */
int in, out /* from 0 to LENGTH */
/* circular buffer implementation */
int empty(void) { return (in == out); }
int full(void) {
    return (((in+1) % LENGTH) == out);
}
/* assume the following are defined */
void produce(item_t *next);
void consume(const item_t *next);
void copy_item(item_t *dst, const item_t *src);
```

Circular Buffer Producer-Consumer

```
void producer(void) {
    item_t next;
    while (1) {
        produce(&next);
        while (full()) /* wait */;
        copy_item(&(buffer[in]), &next);
        in = (in+1) % LENGTH; } }
```

```
void consumer(void) {
    item_t next;
    while (1) {
        while (empty()) /* wait */;
        copy_item(&next, &(buffer[out]));
        out = (out+1) % LENGTH;
        consume(&next); } }
```

Critical Sections

- parts of the program in which shared variables are manipulated
- mutual exclusion is used to avoid races
- a variety of algorithms and hardware support
- conditions for a good solution
 - no assumption about speed or number of CPUs
 - no two processes can simultaneously be inside their critical section (**exclusion**)
 - no process running outside its critical section may block other processes and the selection of who enters can't be postponed indefinitely (**progress**)
 - no process should have to wait forever to enter its critical section (**bounded waiting** or **no starvation**)

Mutual Exclusion with Busy Waiting

- brute force solution: disable interrupts
 - cannot be used in userspace
 - does not work with more than one CPU
- lock variables do not prevent races

```
...  
while (lock == 1); /* busy-wait */  
lock = 1;  
critical_section();  
lock = 0;  
noncritical_section();  
...
```

Busy Waiting: Strict Alternation

process 0:

```
while (1) {  
    while (turn != 0) /* wait */;  
    critical_section();  
    turn = 1;  
    noncritical_section(); }  
}
```

process 1:

```
while (1) {  
    while (turn != 1) /* wait */;  
    critical_section();  
    turn = 0;  
    noncritical_section(); }  
}
```

if one of the processes is slower than the other the **progress** condition is violated

Busy Waiting: Peterson's Solution

```
#define N 2
int turn, interested[N]

void enter_critical_region(int process) {
    int other = 1 - process;
    interested[process] = TRUE;
    turn = other;
    while (turn == other &&
           interested[other] == TRUE);
}

void leave_critical_region(int process) {
    interested[process] = FALSE;
}
```

Hardware Support: Test and Set Lock

- **atomic** TSL instruction
 - `tsl register,lock`
 - copies `lock` to `register` and stores a non-zero value in `lock`
- fictitious assembly code:

```
enter_region:
    tsl register,lock // test and set to 1
    cmp register,#0  // is the lock zero?
    jnz enter_region // loop if non-zero
    ret              // return to caller

leave_region:
    mov lock,#0      // set lock to zero
    ret              // return to caller
```


Synchronization Hazards

- busy-waiting wastes CPU cycles
- race conditions — see above
- priority inversion
 - a low priority process L grabs a lock and enters a critical section
 - a high priority process H busy-waits on the lock
 - a medium priority process M grabs the CPU while L is in the critical section
 - net result: M runs most of the time, sometimes L runs, H is stuck...
- deadlocks...

Deadlocks

Process 1:

```
acquire(lock1);  
acquire(lock2);  
...  
release(lock1);  
release(lock2);
```

Process 2:

```
acquire(lock2);  
acquire(lock1);  
...  
release(lock2);  
release(lock1);
```

- necessary conditions for deadlock:
 - at least one exclusive resource is held
 - a process is holding a resource and waiting for another resource held by another process
 - no preemption: resources can only be released voluntarily
 - circular wait

Dealing with Deadlocks

- protocols ensuring the system will never enter a deadlock state
 - deadlock prevention — ensure that (some of) the necessary conditions do not hold
 - deadlock avoidance — require additional info about resource requests, analyze dynamically
- detecting and recovering from deadlocks
 - may involve killing some processes
- ignore, pretend deadlocks do not exist
 - this is what most OS do, including UNIX/Linux
 - OS is about mechanisms, not policies
 - users' responsibility

Avoiding Busy-Wait: Sleep/Wakeup I

```
#define N 100    /* buffer size */
int count = 0;  /* tracks items in buffer */

void producer(void)
{
    item_t item;
    while (1) {
        produce_item(&item);
        if (count == N) sleep(); /* buffer full */
        put_item(&item);
        count++;
        if (count == 1) wakeup(consumer);
    }
}
```

Avoiding Busy-Wait: Sleep/Wakeup II

```
void consumer(void)
{
    item_t item;
    while (1) {
        if (count == 0) sleep(); /*buffer empty*/
        get_item(&item); /* removes from buffer*/
        count--;
        if (count == N-1) wakeup(producer);
        consume_item(&item);
    }
}
```

● can you see a race here?

Sleep/Wakeup Synchronization

- race condition on `count` — “lost wakeup”
 - the buffer is empty — `consumer` reads `count == 0`
 - scheduler switches to `producer`
 - `producer` puts an item into the buffer, increments `count`, wakes `consumer`
 - `consumer` thinks `count == 0` and goes to sleep!
 - `producer` fills the buffer and goes to sleep, too!
- quick fix — “wakeup waiting bit”
 - set when wakeup is sent to a process that is not sleeping
 - `sleep()` will test the bit, turn it off, remain awake
 - for more than 2 processes more than 1 bit is needed, the problem exists in principle

Semaphores

- Dijkstra (1965) — generalization of sleep/wakeup
- two **atomic** operations — `up()` and `down()` (P and V, according to Dijkstra)
- a counter that controls a shared resource
- `down()` checks the value
 - if positive, the process can use the resource, the semaphore is decremented (indicates that the process is using a “unit” of the resource)
 - if 0, the process goes to sleep
- when the process is done with the resource, `up()` increments the value, processes waiting for the semaphore are awakened
- binary semaphores — 0 or 1

Semaphore Implementation

- `up()` and `down()` must be atomic — kernel support is needed
- typically implemented via system calls that disable interrupts briefly
- for multiple CPUs a (busy-wait) lock is needed to ensure that only one CPU at a time can access the semaphore — `TSL` will help here
- note that this busy-wait is only for the duration of `up()` and `down()`, **not** for the duration of the critical section
- POSIX and SysV semaphores — the latter are much more complicated (more details during the drill session)

Semaphore Usage

- mutual exclusion
 - guarantees that protected resource will not be corrupted by simultaneous access
- synchronization
 - guarantees that certain events will or will not occur
 - producer stops running when the buffer is full
 - consumer stops running when the buffer is empty
- very important: as any lock, semaphores protect data, not code!
 - never ask what parts of code are critical — always ask what data items must be protected or synchronized, and the critical sections will become obvious

Producer/Consumer with Semaphores

```
semaphore mutex = 1, empty = N, full = 0;
```

```
void producer(void)
{
    item_t item;
    while (1) {
        produce(&item);
        down(&empty);
        down(&mutex);
        put_item(&item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer(void)
{
    item_t item;
    while (1) {
        down(&full);
        down(&mutex);
        get_item(&item);
        up(&mutex);
        up(&empty);
        consume(&item);
    }
}
```

Mutexes

- a variable with two states: locked (0) and unlocked (1)
- basic operations
 - `mutex_lock(&mutex)`
 - `mutex_unlock(&mutex)`
- `mutex_lock()` is like `enter_critical_region()`
 - but does not busy-wait
 - if it fails to acquire the mutex it gives up the CPU (using `sched_yield(2)` or similar)
- featuritis: `mutex_trylock(&mutex)`
 - acquires the mutex or returns an error code
 - lets the caller decide whether to yield the CPU or do something else

Semaphore and Mutex Hazards

- both semaphores and mutexes are very prone to programming errors
- many subtle timing issues
- very difficult to avoid
 - enough for one process to be buggy, and everybody suffers
- very difficult to debug (“heisenbugs”)
- some trivial examples
 - `up()` called before `down()` — no exclusion
 - two `down()`'s — deadlock
 - either `up()` or `down()` is not called

Reader/Writer Problem

- a multiprocess (multithreaded) application where processes can read and write the same data
 - several can read simultaneously
 - only one can write at a time
- practical importance: transaction systems, filesystems, etc.
- two types of locks: **shared** for readers, **exclusive** for writers
 - when an exclusive lock is held, no one else can acquire access in any form
 - when a shared lock is held, others can acquire the shared lock, but no one can acquire an exclusive lock

Reader/Writer Flavours

- reader preferred
 - if a reader runs and no one is currently writing, the reader proceeds even if there are writers waiting for access
- writer preferred
 - if a new writer runs in can proceed as soon as possible
- both solutions suffer from possible starvation

Synchronization Performance

- busy-waiting (“spinning” — we’ll discuss spinlocks later)
 - wastes cycles, but if the resource is available or will be available soon avoids a context switch
- blocking (semaphores and mutexes)
 - let other processes run while we wait
 - incurs the overhead of one or more context switches
- hybrid, “adaptive mutexes”
 - start spinning, if the resource is not available spin for a limited time, then go to sleep
 - check if the thread holding the lock is running — it may be likely to release the lock soon (never on UP!)
- reader/writer locks are efficient if reads are much more frequent than writes

Scheduling: Basic Concepts

- all resources, not just CPU, are scheduled
- a process will run happily until it has to wait for I/O
 - we cannot allow the CPU to sit idle
 - multiprogramming: keep several processes in memory, when one has to wait switch to another one, increasing utilization
- CPU “bursts” — periods of CPU activity between I/O waits
 - many more short bursts than long ones
 - I/O-bound processes — lots of short CPU bursts
 - CPU-bound process — a few long CPU bursts
- the (short-term) scheduler selects a process to run from a ready queue (not necessarily a FIFO)

Scheduling Policies: Preemption

- scheduling decisions
 - running → waiting (I/O request, `wait()`)
 - running → ready (interrupt)
 - waiting → ready (completion of I/O)
 - termination
- when a process switches to the “ready” state we need to decide whether to “preempt”
- non-preemptive scheduling — the running process keeps the CPU until it waits or terminates
 - a must for certain hardware (e.g., no timers)
- preemptive scheduling — the CPU may be taken away
 - a must for real-time, high performance computing

Preemption Tradeoffs

- need to protect shared data
 - a process is updating a shared item, is preempted, another process tries to read the inconsistent data
 - similar to multiprocessor systems — discuss later
- kernel preemption
 - the kernel may be updating critical data (e.g., I/O queues) on behalf of a process (during a system call)
 - if preempted, may need to read/modify the same data on behalf of another process — chaos
 - OS may wait for the system call to complete (or an I/O block) before switching context — simpler
 - need to preempt for real time, high performance
 - code affected by interrupts cannot be preempted